Consensus Across Continents

Benjamin Bengfort, Rebecca Bilbro, Pete Keleher
Department of Computer Science
University of Maryland, College Park, MD, USA
{bengfort,rbilbro,keleher}@cs.umd.edu

***Abstract*—Distributing data storage systems across wide geographic areas provides resilience to catastrophic failure and improves performance by localizing user access. However, as network distance increases, the impact of failure modes such as partitions and communication variability pose significant challenges to coordination that impair strong consistency, particularly when systems scale beyond a handful of replicas. In order to balance consistency and performance in a multi-region context, geo-distributed consensus must be flexible, adapting to changing network conditions and user behavior.**

In this paper we introduce Alia, a hierarchical consensus protocol that implements and extends Vertical Paxos, designed to implement large, strongly-consistent and adaptable geo-replicated consensus groups. Alia splits coordination responsibility across two tiers: a root quorum responsible for safely moving the system through reconfigurations, and subquorums that manage accesses. Subquorums intersect with the root quorum using a novel method, delegated voting, which ensures that all replicas participate in both consensus tiers and provide transparent, linearizable guarantees across the entire system. This design ensures Alia can optimize throughput and availability by flexibly changing its configuration in real-time to meet demand without sacrificing consistency.

***Index Terms*—hierarchical consensus, geographic replication, delegated voting, strong consistency**

I. Introduction

Geographically distributed data systems now commonly span continents and oceans. These systems leverage data centers newly available around the globe, increasing local performance by minimizing network distance between users and replicas, and offering data recovery in the face of catastrophes such as floods or earthquakes. Specialized, high-availability data systems [3], [4], [16] have maximized throughput across the wide area, driving interest in geo-replicated systems and making truly international applications increasingly feasible. To generalize geographically distributed systems, managed replicated data services [1], [10], [30] that can provide strong consistency semantics have risen to prominence. However, the solutions introduced by these new systems and services require specialized hardware and engineering involving multiple independent subsystems with different failure modes, which, while providing strong consistency to application developers, do so by hiding both replication and infrastructure complexity.

Unfortunately, this complexity is increasingly necessary for modern application development. Traditional monolithic applications are being replaced by microservice architectures and cloud-native service meshes [17] that make infrastructure directly visible to applications. Developers in turn leverage this visibility to scale applications, for instance using service meshes to maintain and optimize service-specific communication, minimize downtime, localize data to users, and improve system flexibility. This visibility is also critical to meeting the demands of increasing privacy regulations, which require applications developers to finely control data placement based upon differing legal obligations of the locales of users [27]. Thus, while managed data services provide strong consistency, their rigidity and opaqueness are not flexible enough for developers who require strong consistency at a higher level of the application stack.

This paper presents a new protocol, *hierarchical consensus*, motivated by the need for a simpler and more general-purpose approach to building large, geographically replicated systems. The engineering-based solutions of managed geo-distributed data services rely on multiple, independent microservices and quorums together with expensive data-center hardware to synchronize time, allocate locks, manage transactions, and recover from failure. We instead propose a *single, system-wide consensus protocol* that coordinates both replica placement and data accesses. By ensuring that all coordination occurs through a single consensus activity rather than a fleet of small, independent quorums, it is easier to reason about the consistency of the system even in a network environment prone to correlated failures, partitions, and variable latency. This single source of coordination then frees the system to adapt to changes in access patterns, configure to maximize throughput, specify data placement rules, and ensure straightforward system maintenance.

In order to achieve this, a new consensus protocol that can scale beyond a handful of replicas is required. Distributed consensus, canonically represented by Paxos [19] and its performance optimizing variants [6], [8], [20], [21], primarily considers safety in the case of one or two fail-stop node failures. Recent research has explored the problem of geo-distributed consensus [24], [25], but primarily considers the problem of high-latency links. However, geo-replication implies scale. Services running around the globe require dozens if not hundreds of replicas and introduce new failure modes such as network partitions, where sections of the system operate independently without fail-stop failure, and highly variable latency that inhibit quorum progress. To scale systems beyond a handful of replicas, current systems [10], [12], [18], [28] use Paxos as a component, instantiated across multiple transactions, shards, or tablets to manage small subsystems independently, making it difficult to reason about consistency.

Hierarchical consensus introduces a novel approach to scale

consensus beyond a handful of nodes, wherein the consensus problem is effectively decomposed into process units. This is achieved with a multi-group coordination protocol that configures and mediates subquorums through a root quorum. The root quorum guarantees correctness by pivoting the system through *reconfigurations* that place replicas into subquorums, mapping them to partitions of the object namespace. Each subquorum serializes accesses to its mapped objects using provenly safe algorithms, placed to maximize throughput or durability. To ensure that all system-wide consensus decisions are totally ordered with respect to changes in reconfiguration, all subquorums intersect the root quorum in a hierarchy of quorums.

The root quorum is composed of all replicas in the system. Though decisions made by the root quorum are rare with respect to data accesses, we introduce *delegated voting* to optimize quorum decisions at the root. Much of the system's complexity comes from handshaking between the root quorum and subquorums during reconfiguration. These handshakes are made easier and far more efficient by using *fuzzy transitions*, which allow individual subquorums to move through reconfiguration at their own pace without impeding progress. Finally, subquorum consensus can be optimized for policy-driven *data placement*, allowing objects that require more throughput to use leader-oriented consensus whereas objects that require stronger durability can be replicated across data centers using optimistic fast-path consensus.

We validate our approach by implementing hierarchical consensus in Alia, a linearizable object store explicitly intended to run with many replicas, geo-replicated across heterogenous networks and devices. The resulting system is local, in that replicas serving clients can be located near them. The system is fast because individual operations are served by a small group of replicas regardless of the size of the total system. The system is nimble in that it it can dynamically reconfigure the number, membership, and responsibilities of the subquorums in response to failures, phase changes in the driving applications or policy requirements for data placement and durability. Finally, the system is consistent, supporting the strongest form of per-object consistency without relying on special-purpose hardware. We demonstrate its advantages through an implementation scaling to hundreds of replicas across more than a dozen availability zones around the world using Amazon EC2.

II. Background

Distributed consensus algorithms ensure that multiple replicas maintain an identical state by applying commands in the same order, ensuring a consistent, externalizable view of the system when any replica is queried. Most of these algorithms are based on Paxos [19], which ensures that a distributed log of ordered commands is maintained even if f replicas fail-stop. This is achieved with quorums of $2f+1$ replicas that write an entry into the log using a 2 phase balloting process: `PREPARE` and `ACCEPT`. The `PREPARE` phase is designed to nominate an open slot in the log at position i to place the command and to detect any conflicting commands that may already exist in that slot; the `ACCEPT` phase commits the command in that slot. In both phases a majority of replicas must respond for progress to be made.

There are two primary optimizations to the basic consensus algorithm: leader election and fast path execution. Leader-oriented consensus protocols elide the `PREPARE` phase by specifically selecting a replica that has sole responsibility of slot nomination. Some variants such as Raft [26] elect a leader that can nominate all slots for a time-limited duration, where as others like Mencius [24] use a round-robin approach to assigning leadership. Fast path execution such as Fast Paxos [21] and EPaxos [25] optimistically push through commits on the `PREPARE` phase and use conflict detection mechanisms to determine if a slow path `ACCEPT` is required. Both of these optimizations are designed to reduce communication rounds in the common case, while still maintaining safety when f replicas fail, however none of these optimizations describe how to scale consensus beyond $2f+1$ replicas.

Vertical Paxos [22], [23] describes how to scale consensus groups by allowing an auxiliary master quorum to execute safe reconfiguration directives in the middle of consensus decisions. The replicated state machine process is extended from a single log to a grid where commands are placed both horizontally as a sequence of consensus instances (configurations) and vertically as increasing ballot numbers (log indices). Because consensus instances are disjoint inside of a single configuration, Vertical Paxos creates a total ordering of commands that is guaranteed to bring the system to an identical state no matter the point in execution. By allowing multiple, active consensus instances, Vertical Paxos decouples command execution and state transfer from reconfiguration, allowing the system to arbitrarily scale. While Vertical Paxos considers scaling consensus and other algorithms such as EPaxos and Mencius consider geo-distributed consensus, no consensus protocol tackles scaling consensus to hundreds of nodes across the wide area.

III. Hierarchical Consensus

Hierarchical consensus and Alia are an extension and implementation of Vertical Paxos that are specifically designed for scalable, geo-distributed consensus. Like Vertical Paxos, hierarchical consensus (HC) organizes replicas into two tiers of quorums, each responsible for fundamentally different decisions, as shown in Figure 1. The lower tier consists of multiple independent subquorums, each committing data access operations to local shared logs. The upper, root quorum, consists of subquorum peers, usually their leaders, delegated to represent the subquorum and hot spares in root elections and reconfiguration commits. HC's main function is to export a linearizable abstraction of shared accesses to some underlying substrate, such as a distributed object store or file system. We assume that nodes hosting object stores, applications, and HC are frequently co-located across the wide area and introduce our object store implementation, Alia, in Section V.

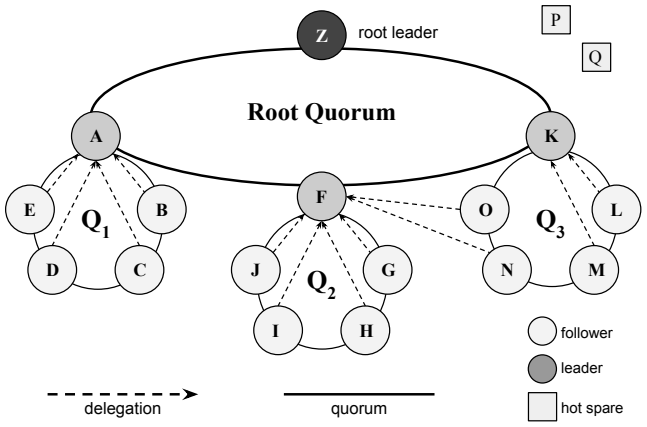

Fig. 1. Hierarchical consensus is an extension of Vertical Paxos wherein the root quorum moves the system through reconfigurations, and intersecting subquorums manage local data accesses.

The root quorum's primary responsibility is to manage the configuration of P distinct processes, each of which maintain their own local state and can handle access requests from users to objects represented by a namespace, N. It does this by mapping processes to subquorums and mapping subquorums to partitions of the namespace, allocating all extra processes as *hot spares*, which stand by to be placed in a subquorum on individual process failure. Each distinct mapping of the namespace to replicas is a single configuration and is represented by an *epoch*, e, a monotonically increasing index that describes the horizontal position of the Vertical Paxos grid, and is used by the system to guarantee safety. The set of subquorums specified by each epoch is represented by Q_e such that an individual subquorum is identified as $q_{i,e}$.

The root quorum partitions (shards) the namespace across Q_e by mapping a subset of the namespace across the subquorums, $n_i \mapsto q_{i,e}$, ensuring that each shard of the namespace is a disjoint subset such that $\forall n_i \in N_e \; \exists! q_{i,e} \mapsto n_i$. The intent of subquorum localization is ensure that the *domain* of a client, the portion of the namespace it accesses, is entirely within the scope of a local, or nearby, subquorum. To the extent that this is true across the entire system, each client interacts with only one subquorum, and subquorums do not interact at all during execution of a single epoch. This *siloing* of client accesses simplifies implementation of strong consistency guarantees and allows better performance at the cost of restricting multi-object transactions. We use agility to attempt to get this, but allow client-side multi-object transactions.

The root quorum is a consensus group consisting of delegates (usually subquorum leaders) whose primary consensus operation is to commit new epochs. As subquorums transition to new epochs, they handoff their current state to the subquorum (s) newly responsible for managing that portion of the namespace, ensuring that no portion of the namespace can be accessed concurrently in two different epochs. For understandability we describe the HC in leader-oriented terms, though there is no requirement for either the root quorum or the subquorums to implement leader-oriented consensus. Further, we propose that subquorums should be assigned the consensus algorithm that best optimizes for the durability and throughput requirements of the shards they manage. While the root quorum is composed of all replicas in the system, only a subset of replicas actively participates in root quorum decision making, in the common case. Using these mechanisms, hierarchical consensus prioritizes flexibility and transparency, ensuring the system is safe while able to make progress and can be managed easily while scaled to very large consensus groups across continents and oceans.

A. Delegated Voting

From a logical perspective, the root quorum's membership is the set of all system replicas, at all times. This ensures that all subquorums intersect with the root quorum such that all commands are totally ordered, and to improve the overall fault tolerance and flexibility of the system, eliminating the root quorum as a potential bottleneck. However, running consensus elections across large systems is inefficient in the best of cases, and prohibitively slow in a geo-replicated environment. Root quorum decision-making is kept tractable by having replicas *delegate* their votes, usually to their local leaders, for a finite duration. With leader delegation, the root membership effectively consists of the set of subquorum leaders in the ideal case. Because subquorums themselves are geographically distributed and placed to optimize system specific policies, the root quorum represents a simplified snapshot of the overall operation and as the network environment and access patterns change, so too does the root quorum. To simplify the discussion of delegated voting, we assume a leader-oriented consensus protocol that operates similarly to Raft, using heartbeats and timeouts to detect failure.

The root quorum elects a leader for a root term, r and broadcasts this term along with routine heartbeats that keep the leader term alive. Each replica has a delegated term, d, on root heartbeat, if $r > d$, then the replica sets $d = r$ and marks its vote as no longer delegated. If the replica is not delegated it can issue a delegate request to local replicas for a delegate term, $d_t = d + j$ where j defines how many root elections the delegation will survive. Usually, $j = 1$, but can be increased if intermittent failure in the root quorum is expected (to allow the delegates to smoothly transition to new leaders without re-delegation). When a delegate request is received, the replica can delegate iff $d_t > d$ and marks itself as delegated, otherwise it reports the current term to the delegate candidate. A *delegate* is any replica that is able to vote in root term r, e.g. any replica whose $d > r$.

During a consensus operation in the root quorum for either a root leader election or an epoch change, the root leader or candidate broadcasts a request to *all replicas* with its current term. Replicas only reply to the vote if $d \geq r$ and they have not delegated their vote. We consider it possible, depending on the implementation, that duplicate votes might be received, therefore to ensure safety, the root leader or candidate must account for which replicas have voted, keeping only the votes

for the highest delegated term. This is true even for hot spares, which are not currently in any subquorum. Delegates reply with the unique ids of the replicas they represent so that root consensus decisions are still made using a majority of all system replicas. This is correct because vote requests now reach all replicas, and because replicas whose votes have been delegated merely ignore the request. We argue that it is also efficient, as a commit's efficiency depends only on receipt of a majority of the votes.

Delegation ensures that root quorum membership is always the entire system and remains unchanged over subquorum leader elections and even reconfiguration. Delegation is essentially a way to optimistically shortcut contacting every replica for each decision. Large consensus groups are generally slow, not just because of communication latency, but because large groups in a heterogeneous setting are more likely to include replicas on very slow hosts or networks. In the usual case for our protocol, the root leader still only needs to wait for votes from the subquorum leaders. Leaders are generally those that respond more quickly to timeouts, so the speed of root quorum operations is unchanged. Note that because the root term only increments when the root leader changes, epoch decisions are the commonly delegated votes.

Delegation can also be seen as a `PRE-PREPARE` phase before the root leader election (e.g. the root quorum `PREPARE`). If a root election timeout expires, **only a delegate can become a root candidate** (including replicas that vote only for themselves). This means that at least one root election will occur with delegation, prompting re-delegation requests that are generally far in advance of new elections. In normal operation, with limited failure, all root decisions are delegated. However, if the delegates representing a majority of the system fail, so long as there is at least one live delegate, the root term will increase until all delegations are busted, causing all replicas to re-delegate their votes to a live replica, or to simply vote for themselves. There is one critical edge case, all delegates failing simultaneously, which we discuss in Section IV-B.

The optimal root quorum configuration is to have all subquorum leaders as delegates with hot spares delegated evenly to local delegates. This configuration balances the fault tolerance of the root quorum with the throughput of decision making, however this configuration is not guaranteed by the protocol described above. Our approach employs heuristic mechanisms such as network distance or maximum delegation to ensure the efficiency of delegated voting and to limit delegation inside of single regions only. The root quorum leader can simplify this process by identifying replicas that have previously been highly available members of the root quorum. If the root quorum leader identifies a poor delegation state, it simply initiates a root leader election for itself, incrementing r, and causing replicas to re-delegate after the election. On epoch change, the root leader can hint to the reconfigured subquorums the best replicas to delegate to if they are not already delegated for the epoch. If no hints are provided, then replica followers generally delegate their vote to the term 1 leader and hot spares to the closest subquorum leader. Generally, the reconfigurations that occur during epoch changes allow the root quorum to move the system to an optimal state given current network conditions.

B. Reconfiguration

Every epoch represents a new configuration of the system as designated by the root leader. Efficient reconfiguration ensures that the system is both dynamic, responding both to failures and changing usage patterns, and minimizes coordination by colocating related objects. An epoch change is initiated by the root leader in response to one of several events, including:

- notification of failed replicas
- a namespace repartition request to optimize data accesses or minimize conflicts
- changing network conditions that suggest re-assignment to improve subquorum throughput
- application-initiated reconfigurations to localize data, increase durability, or enforce policies

The root leader transitions to a new epoch through the normal commit phase in the root quorum. The command proposed by the leader is an enumeration of the new subquorum partition, namespace partition, and assignment of namespace portions to specific subquorums. The announcement may also include initial leaders for each subquorum, with the usual rules for leader election applying otherwise, or if the assigned leader is unresponsive. Upon commit, the operation serves as an *announcement* to subquorum leaders. Subquorum leaders repeat the announcement locally, disseminating full knowledge of the new system configuration, and eventually transition to the new epoch by committing an `epoch-change` operation locally.

The epoch change is lightweight for subquorums that are not directly affected by the overarching reconfiguration. If a subquorum is being changed or dissolved, however, the *epoch-change* commitment becomes a tombstone written to the logs of all local replicas. No further operations will be committed by that version of the subgroup, and the local shared log is archived and then truncated. Truncation is necessary to guarantee a consistent view of the log within a subquorum, as peers may have been part of different subquorums, and thus have different logs, during the last epoch. Replicas then begin participating in their new subquorum instantiation. In the common case where a subquorum's membership remains unchanged across the transition, an `epoch-change` may still require additional mechanism because of changes in namespace responsibility.

C. Data Placement

Reconfiguration by the root quorum not only allows the system to transparently scale consensus operations, but also provides applications the ability to directly specify how and where consensus operations should be placed. Although epoch changes are relatively heavy weight, the design of HC allows them to be routine. Most systems are optimized for only one type of data placement, however real-world applications vary

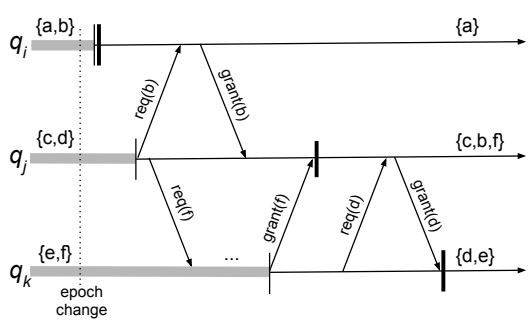

Fig. 2. Fuzzy transitions allow the system to safely make progress unimpeded by leadership changes in either the root quorum or subquorum.

in their needs for availability and durability, often on a per-object basis. Applications may also observe markedly different access patterns that change during the course of operation, including:

- single ownership: objects are accessed in one region and do not migrate.
- revolving access: objects are accessed from a primary region that changes over time.
- conflicting access: objects are continuously accessed simultaneously from multiple regions.

These access patterns imply not only the optimal placement of replicas in specific geographic regions, but also the consensus protocol that should be used to maximize throughput. For example, if commit latency for significant numbers of accesses to a small set of objects outweighs durability requirements, a subquorum can be placed in a single region using Raft, which serializes all accesses through a single leader and is immune to conflicting reads and writes. To increase durability, data can be placed with a leader and primary backup in a single region, and a secondary backup in a remote region (it is important with this scheme to modify the election rules of Raft to decrease the probability that the secondary backup is elected leader). To handle conflicting accesses across regions, EPaxos or Mencius can be used to optimistically serialize proposals across the wide area. HC requires no special modifications to subquorum operation, only requiring that subquorums intersect with the root quorum and that delegation and transitions are implemented correctly. As a result, the root quorum can specify not only the placement of namespace shards with the replica, but also the consensus algorithm that governs them. We envision the root quorum as a distributed system management framework that is not only able to apply policy changes efficiently, but also to monitor real-time performance and to make adaptations on demand. Adaptations are the key to the protocols flexibility but also ensure that the system is lightweight enough to be managed at a global scale without significant resources.

D. Fuzzy Transitions

Epoch handshakes are required whenever the namespace-to-subquorum mapping changes across an epoch boundary. HC separates epoch transition announcements in the root quorum from implementation in subquorums. Epoch transitions are termed *fuzzy* because subquorums need not all transition synchronously. There are many reasons why a subquorum might be slow. Communication delays and partitions might delay notification or temporary failures might block local commits. A subquorum might also delay transitioning to allow a local burst of activity to cease such as currently running transactions[1]. Safety is guaranteed by tracking subquorum dependencies across the epoch boundary.

Figure 2 shows an epoch transition where the scopes of q_i, q_j, and q_k change across the transition to epoch e as follows:

$$q_{i,e-1} = n_a, n_b \longrightarrow q_{i,e} = n_a$$
$$q_{j,e-1} = n_c, n_d \longrightarrow q_{j,e} = n_c, n_d, n_f$$
$$q_{k,e-1} = n_e, n_f \longrightarrow q_{k,e} = n_d, n_e$$

All three subquorums learn of the epoch change at the same time, but become ready with varying delays. These delays could be because of network lags or ongoing local activity. Subquorum q_i gains no new objects across the transition and moves immediately to the new epoch. Subquorum q_j's readiness is slower, but then it sends requests to the owners of both the new objects it acquires in the new epoch. Though q_i responds immediately, q_k delays its response until locally operations conclude. Once both handshakes are received, q_j moves into the new epoch, and q_k later follows suit.

These bilateral handshakes allow an epoch change to be implemented incrementally, eliminating the need for lockstep synchronization across the entire system. This flexibility is key to coping with partitions and varying connectivity in the wide area. However, this piecewise transition, in combination with subquorum re-definition and configuration at epoch changes, also means that individual replicas *may be part of multiple subquorums at a time*.

This overlap is possible because replicas may be mapped to distinct subgroups from one epoch to the next. Consider q_k in Figure 2 again. A single replica process, p, may be remapped from subquorum $q_{k,e-1}$ to subquorum $q_{i,e}$ across the transition. Subquorum $q_{k,e-1}$ is late to transition, but $q_{i,e}$ begins the new epoch almost immediately. Requiring p to participate in a single subquorum at a time would potentially delay $q_{i,e}$'s transition and impose artificial synchronicity constraints on the system. One of the many changes we made in the base Raft protocol is to allow a replica to have multiple distinct shared logs. Smaller changes concern the mapping of requests and responses to the appropriate consensus group.

[1]The HC protocol discussed in this paper does not currently support transactions.

IV. Safety and Fault Tolerance

We assert that consensus at the leaf quorums is correct and safe because decisions are implemented using well-known consensus approaches. Hierarchical consensus therefore has to demonstrate linearizable correctness and safety between subquorums for a single epoch and between epochs during transitions. Briefly, linearizability requires external observers to view operations to objects as instantaneous events. Within an epoch, subquorum leaders serially order local accesses, thereby guaranteeing linearizability for all replicas in that quorum. All accesses are redirected to the subquorum that governs that part of the namespace, therefore inside of an epoch, external observers can only observe a single total ordering to each object.

Epoch transitions raise the possibility of portions of the namespace being re-assigned from one subquorum to another, with each subquorum making the transition independently. Correctness is guaranteed by an invariant requiring subquorums to delay serving newly acquired portions of the namespace until after completing all appropriate handshakes. Between epochs, it is possible to construct a single total ordering of all accesses using a log grid such that epochs represent the horizontal index and each subquorum log represents the vertical index as described in Vertical Paxos [22]. Serializing accesses across multiple objects can be accomplished by ensuring that those objects are placed in the same namespace. Therefore to prove safety, hierarchical consensus must demonstrate that it operates correctly when replica processes fail.

A. Failures

During failure-free execution, the root quorum partitions the system into disjoint subquorums, assigns *subquorum leaders*, and assigns shards of the namespace to subquorums. Each subquorum coordinates and responds to accesses for objects in its assigned shard. We define the system's *safety* property as guaranteeing that non-linearizable (or non-sequentially-consistent) event orderings can never be observed. We define the system's *progress* property as the system having enough live replicas to commit votes or operations in the root quorum.

The system can suffer several types of failures, as shown in Table IV-A. We assume that each of these failure types are detected by the absence of communication and that replicas may automatically rejoin the system once communication is re-established. Failures of subquorum and root quorum leaders are handled through the normal consensus mechanisms. Failures of subquorum peers are handled by the local leader petitioning the root quorum to re-configure the subquorum in the next epoch. Failure of a root quorum peer is the failure of subquorum leader, which is handled as above. Root quorum heartbeats help inform other replicas of leadership changes, potentially necessary when individual subquorums break down.

HC's structure means that some faults are more important than others. Proper operation of the root quorum requires the delegates representing the majority of replicas to be non-faulty. We consider the case where all replicas are assigned to a subquorum and the delegates are the subquorum leaders. Given a system with $2m + 1$ subquorums, each of $2n + 1$ replicas, the entire system's progress can be halted with as few as $(m + 1)(n + 1)$ well-chosen failures. Therefore, in worst case, the system can only tolerate:

$$f_{worst} = mn + m + n$$

failures and still make progress. At maximum, HC's basic protocol can tolerate up to:

$$f_{best} = (m + 1) * n + m * (2n + 1) = 3mn + m + n$$

failures. As an example, a $m = 12$ with $n = 2$ system is composed of 25 total replicas that operate subquorums of 5 replicas each and can tolerate at least 8 and up to 16 failures. An $m = 10$, $n = 1$ system can tolerate at least 7, and a maximum of 12, failures out of 21 total replicas and subquorums of 3 replicas. Individual subquorums might still be able to perform local operations despite an impasse at the global level. Total subquorum failure can temporarily cause a portion of the namespace to be unserved. However, the root quorum eventually times out and elects a new leader, which busts the delegations and allows a reconfiguration that will allow the system to operate normally.

B. Assassination

Singleton consensus protocols, including Raft and EPaxos, can tolerate just under half of the entire system failing. As described above, HC's structure makes it more vulnerable to clustered failures. Therefore we define a *disaster timeout*, which uses direct consensus decision among all system replicas to tolerate any f replicas failing out of $2f+1$ total replicas in the system. In particular, we are concerned about the edge case where all delgates simultaneously fail, an irregular occurrence that eliminates the root leader and all possible root leader candidates such that the root term would never be incremented to bust delegation. Because of the specificity of such a failure, we describe it as *assassination*.

The disaster vote is triggered by the absence of communication from the root leader for a time period significantly longer than it would normaly take to simply elect a new root leader from the available delegates. Any replica may become a root leader candidate if this timeout occurs by incrementing its term for the root quorum quorum such that $r > d$. The key difficulty is in preventing delegated votes and disaster votes from reaching conflicting decisions. Such situations might occur when temporarily unavailable delegates regain connectivity and allow a wedged root quorum to unblock. Meanwhile, a disaster vote might be concurrently underway.

Replica delegations are defined as intervals over specific slots. Using local subquorum slots would fall prey to the above problem, so we define delegations as a small number (often one) of root terms. During failure-free operation, peers delegate to their leaders and are all represented in the next root election or commit. Peers then renew their delegations to their leaders when a new root leader is elected. Consider a subquorum where all peers have delegated votes to their

Failure Type	Response
subquorum peer	request replica repartition from root quorum
subquorum leader	local election, request replacement from root quorum
root leader	root election (with delegations)
delegate	root election(s) until delegations busted
all delegates (assassination)	root election after disaster timeout

leader for the next root slot. If that leader fails, none of the peers will be represented. The first response is initiated when a replica holding delegations (or its own vote) times out waiting for the root heartbeat. That replica increments its own root term, adopts the prior system configuration as its own, and becomes a root candidate. This candidacy fails, as a majority of subquorum leaders, with all of their delegated votes, are gone. Progress is not made until a second root election time out occurs, causing a second incrementing of the root term and causing all delegations to lapse. In our default case where a delegation is for a single root event, this happens after the first root election failure and allows the entire system to vote.

However, if this delegate remains partitioned, it is possible that there is a concurrent vote from a non-delegate candidate after the disaster timeout. Because the subquorums intersect with the root quorum it is impossible for a delegate to achieve a majority while being partitioned, therefore the disaster candidate election will succeed. When the partitioned delegates rejoin the system, their root term will be lower than the term of the disaster candidate's, therefore they will lose their delegations and will rejoin the system as a non-delegate replica.

V. Implementation

Alia implements a replicated key-value store with strong consistency via the hierarchical consensus protocol using Golang and gRPC for communication. Alia is designed as a single process, such that an Alia cluster is composed of P processes of the same type, each of which manage a partial replica of the object space and participate in *both* root consensus and subquorum consensus. Consensus and delegated voting are implemented using modified Raft as the underlying protocol. Each replica implements an event loop that responds to timing events, client requests, and messages from peers. Events may cause the replica to change state, modify a command log, broadcast messages to peers, modify the local store, or respond to a client. Because events are crticial to the correctness and safety of the system, all events are serialized through a single channel so that they are handled sequentially by the primary process.

In addition to the major changes to base Raft such as allowing replicas to be part of multiple subquorums simultaneously and to accept delegated votes, we also made many smaller changes that had pervasive effects. One such change was including the epoch number alongside the term in all log entries and using it as an invariant whether a replica is as up to date as another log. We allowed aggregation of multiple client requests into a single round by subquorum leaders, which helped increase throughput and decrease the number of required messages. We also implemented hierarchical thrifty communications to minimize the number of messages such that root quorum would send messages to delegates first, then broadcast to all replicas if delegate messages were not acknowledged.

VI. Evaluation

HC was designed to adapt both to dynamic workloads as well as variable network conditions. We therefore evaluate HC in three distinct environments: a homogeneous data center, a heterogeneous real-world network, and a globally distributed cloud network. The homogeneous cluster is hosted on Amazon EC2 and includes 26 "t2.medium" instances: dual-core virtual machines running in a single VPC with inter-machine latencies (λ) normally distributed with a mean, $\lambda_\mu = 0.399ms$ and standard deviation, $\lambda_\sigma = 0.216ms$. The heterogeneous cluster (UMD) consists of several local machines distributed across a wide area, with inter-machine latencies ranging from $\lambda_\mu = 2.527ms$, $\lambda_\sigma = 1.147ms$ to $\lambda_\mu = 34.651ms$, $\lambda_\sigma = 37.915ms$. The variability of this network also poses challenges that HC is uniquely suited to handle via root quorum-guided adaptation. We explore two distinct scenarios – sawtooth and repartitioning – using this cluster; all other experiments were run on the EC2 cluster.

In our final experiment, we explore the use of HC in an extremely large, planetary-scale system comprised of 105 replicas in 15 data centers in 5 continents spanning the northern hemisphere and South America. This experiment was also hosted on EC2 "t2.medium" instances in each of the regions available to us at the time of this writing. In this context, reporting average latencies is difficult as inter-region latencies depend more on network distance than can be meaningfully ascribed to a single central tendency.

A. Basic Performance

HC is partially motivated by the need to scale strong consistency to large cluster sizes. We based our work on the assumption that consensus performance decreases as the quorum size increases, which we confirm empirically in Figure 3. This figure shows the maximum throughput against system size for a variety of workloads, up to 120 concurrent clients. A workload consists of one or more clients continuously sending writes of a specific object or objects to the cluster without pause.

Standard consensus algorithms, Raft in particular, scale poorly with uniformly decreasing throughput as nodes are added to the cluster. Commit latency increases with quorum size as the system has to wait for more responses from peers, thereby decreasing overall throughput. Figures 3 and 4 clearly show the multiplicative advantage of HC's hierarchical structure. Note that though HC is not shown to scale linearly

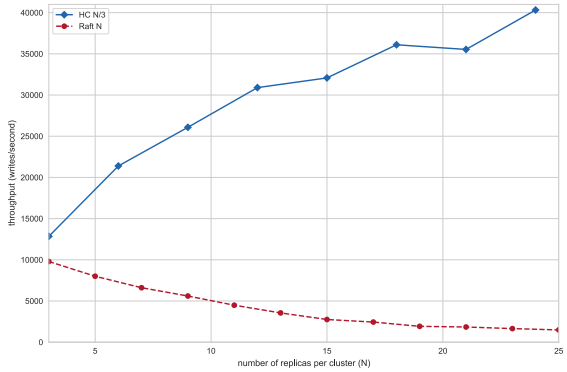

Fig. 3. Throughput increases with larger quorum sizes instead of decreasing.

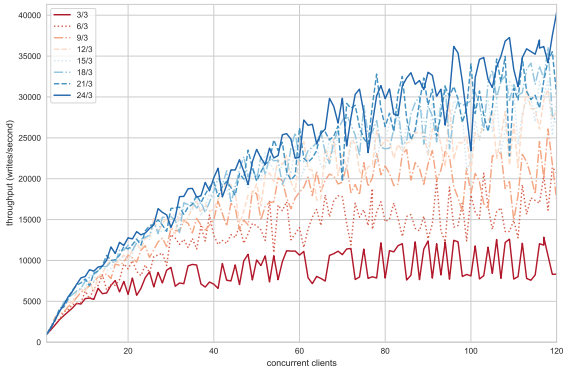

Fig. 4. HC scales to handle workloads without coordination bottlenecks.

in these figures, this is due to performance bottlenecks of the networking implementation in these experiments. In our final experiment, we show linear scaling with our latest implementation of HC.

There are at least two factors limiting the HC throughput shown in our initial experiments. First, the HC subquorums for the larger system sizes are not saturated. A single 3-node subquorum saturates at around 25 clients and this experiment has only about 15 clients per subquorum for the largest cluster size. We ran experiments with 600 clients, saturating all subquorums even in the 24-node case. This throughput peaked at slightly over 50,000 committed writes per second, better but still lower than the linear scaling we had expected.

We think the reason for this ceiling is hinted at by Figure 4. This figure shows increasingly larger variability with increasing system sizes. A more thorough examination of the data shows widely varying performance across individual subquorums in the larger configurations. After instrumenting the experiments to diagnose the problem, we determined it was a bug in the networking code, which we repaired and improved. By aggregating append entries messages from clients while consensus messages were in-flight, we managed to dramatically increase the performance of single quorums and reduce the number of messages sent. This change also had the effect of ensuring that the variability was decreased in our final experiment.

The effect of saturation is also demonstrated in Figure 5, which shows cumulative latency distributions for different system sizes holding the workload (number of concurrent clients) constant. The fastest (24/3) shows nearly 80% of client write requests being serviced in under 2 msec. Larger system sizes are faster because the smaller systems suffer from contention (25 clients can saturate a single subquorum). Because throughput is directly related to commit latency, throughput variability can be mitigated by adding additional subquorums to balance load.

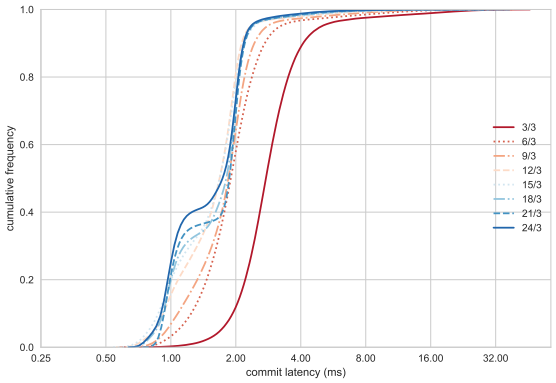

Fig. 5. Cumulative latency of requests decreases with system size.

B. *Adaptability*

Besides pure performance and scaling, HC is also motivated by the need to adapt to varying environmental conditions. In the next set of experiments, we explore two common runtime scenarios that motivate adaptation: shifting client workloads and failures. We show that HC is able to adapt and recover with little loss in performance. These scenarios are shown in Figures 7 and 6 as throughput over time, where vertical dotted lines indicate an epoch change.

The first scenario, described by the time series in Figure 6 shows an HC 3-replica configuration moving through two epoch changes. Each epoch change is triggered by the need to localize objects accessed by clients to nearby subquorums. The scenario shown starts with all clients co-located with the subquorum serving the shard of the namespace they are accessing. However, clients incrementally change their access patterns first to an object located on one remote subquorum, and then to the object owned by the other. In both cases, the root quorum adapts the system by repartitioning the namespace such that the object defining their current focus is served by the co-located subquorum.

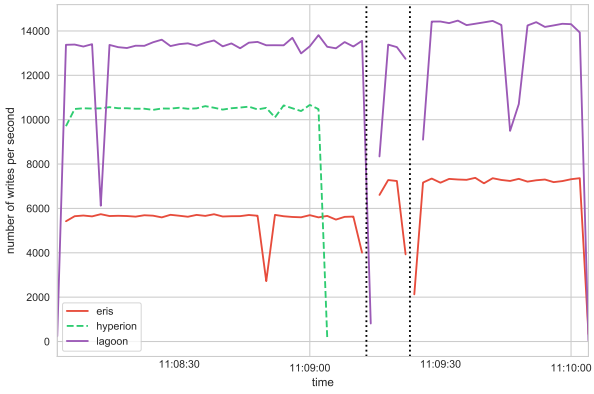

Fig. 6. Reconfiguration to adapt to changing access patterns.

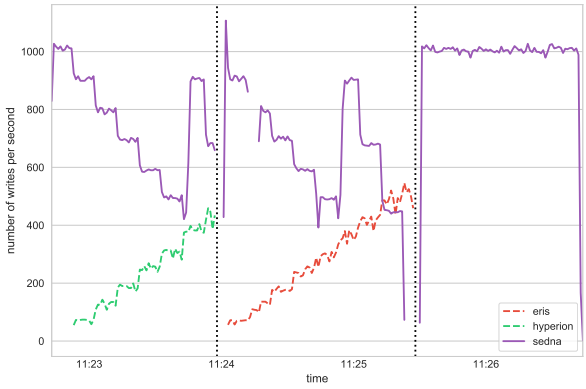

Fig. 7. Reconfiguration to take over from failing subquorums.

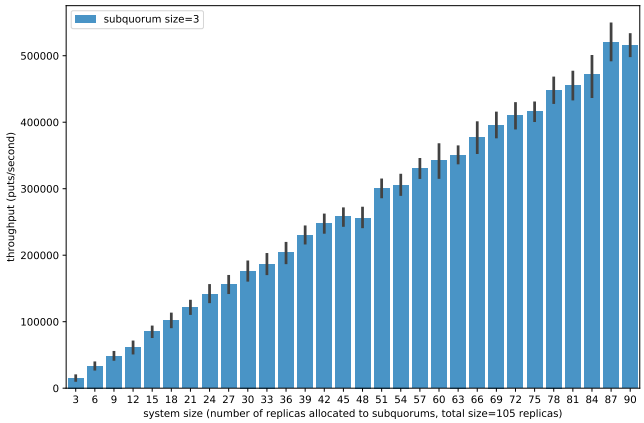

Fig. 8. Consensus scales linearly as the number of replicas increases.

Figure 6 shows a 3-subquorum configuration where one entire subquorum becomes partitioned from the others. After a timeout, the root uses an epoch change to re-allocate the namespace of the partitioned subquorum over the two remaining subquorums. The partitioned subquorum eventually has an heuristic *obligation timeout*, after which the root quorum is not obliged to leave the shard with the current subquorum. The shard may then be re-assigned to any other subquorum. Timeouts are structured such that by the time an obligation timeout fires, the root quorum has already re-mapped that subquorum's shard to other subquorums. As a result, the system is able to recover from the partition as fast as possible. In this figure, the repartition occurs through two epoch changes, the first allocating part of the namespace to the first subquorum, and the second allocating the rest of the namespace to the other. Gaps in the graph are periods where the subquorums are electing local leaders. This may be optimized by having leadership assigned or maintained through root consensus.

C. Planet Scale Consensus

In our final implementation we ran our repaired version of HC at a planetary scale. We created a system with 105 replicas in 15 regions in 5 continents. The system allocated size 3 subquorums round-robin to each region such that the largest system was comprised of 6 subquorums per region with 1 hot-spare per region. Figure 8 shows the global blast throughput of the system, the sum of throughput of client process that fired off 1000 concurrent requests, timing the complete response. To mitigate the effect of global latency, each region ran independent blast clients to its local subquorums, forwarding to remote quorums where necessary. To ensure that the system was fully throttled during the throughput experiment, we timed the clients to execute simultaneously using the AWS Time Sync service to ensure that clocks were within 100 nanoseconds of each other. In these results we show that our HC implementation does indeed scale linearly. Adding more nodes to the system increases the fault tolerance (e.g. by allocating hot s pares) if enough nodes are added to add another subquorum, the capacity of the system to handle client requests is also increased.

VII. Related Work

The principal contribution of this paper, hierarchical consensus, follows from the large body of work on improving throughput in distributed consensus over the Paxos protocol [13], [19], [20], [25], and on Raft [14], [26], which focus primarily on fast vs. slow path consensus, eliding phases with dependency resolution, and load balancing.

Our work is orthogonal to these in that subquorums and the root quorum can be implemented with different underlying consensus algorithms, though the two levels must be integrated quite tightly. Further, HC abstracts reconfiguration away from subquorum consensus, allowing multiple subquorums to move into new configurations and reducing the need for joint consensus [26] and other heavyweight procedures. Finally, its hierarchical nature allows the system to multiplex multiple

consensus instances on disjoint partitions of the object space while still maintaining global consistency guarantees.

The global consistency guarantees of HC are in direct contrast to other systems that scale by exploiting multiple consensus instances [10], [18] on a per-object basis. These systems retain the advantage of small quorum sizes but cannot provide system-wide consistency invariants. Another set of systems uses quorum-based decision-making but relaxes consistency guarantees [9], [11]; others provide no way to pivot the entire system to a new configuration [12]. Chain replication [29] and Vertical Paxos [22], [23] are among approaches that control Paxos instances through other consensus decisions. However, HC differs in the deep integration of the two different levels. Whereas these approaches are top down, HC consensus decisions at the root level replace system configuration at the subquorum level, and vice versa.

Possibly the closest system to HC is Scatter [12], which uses an overlay to organize consistent groups into a ring. Neighbors can join, split, and talk amongst themselves. The bottom-up approach potentially allows scaling to many subquorums, but the lack of central control makes it hard to implement global re-maps beyond the reach of local neighbors. HC ties root quorum and subquorums tightly together, allowing root quorum decisions to completely reconfigure the running system on the fly either on demand or by detecting changes in network conditions.

Recent work has similarly explored a more strategic approach to data placement; Akkio [3], [16], for instance, optimizes shard placement using access latency as the cost function. HC goes a step further, offering not optimization but rather fine grain control over data placement, which allows applications to determine their own use-case specific cost functions and heuristics.

We claim very strong consistency across a large distributed system, similar to Spanner [10]. Spanner provides linearizable transactions through use of special hardware and environments, which are used to tightly synchronize clocks in the distributed setting. Spanner therefore relies on a very specific, curated environment. HC targets a wider range of systems that require cost effective scaling in the data center to rich dynamic environments with heterogeneity on all levels.

Finally, shared logs have proven useful in a number of settings from fault tolerance to correctness guarantees. However, keeping such logs consistent in even a single consensus instance has proven difficult [7], [15]. More recent systems are leveraging hardware support to provide fast access to shared logs [2], [5], [31]. To our knowledge, hierarchical consensus is the first work to propose synchronizing shared logs across multiple discrete consensus instances in the wide area.

VIII. Discussion

Alia takes a different approach to implementing geo-distributed systems, focusing on a system's ability to be *flexible*. Flexibility ensures that the system can balance requirements for throughput and availability while still maintaining the strongest possible consistency semantics. To achieve this, Alia is based on three primary design requirements that inform the rest of the framework.

Requirement 1: Systems should be as fluid as the information they contain. Many systems are optimistic, they assume that conflict is rare and that objects are accessed in standard patterns that change. In our experience, both people and information flows freely therefore a system must accommodate organic and shifting usage patterns; for example a set of objects may primarily be accessed only in daylight, requiring the system to adapt by moving the coordinating replicas to the locales currently in working hours. To accommodate this requirement, Alia is designed to regularly and safely transition through reconfigurations called epoch changes, reallocating replicas into subquorums to manage specific partitions of the namespace. Epoch changes are *fuzzy* to ensure that reconfiguration does not need to be synchronous and hand-offs are optimized through anti-entropy replication of data.

Requirement 2: No partial failures. A system's size should be its advantage – allowing increased throughput with linear scaling, and better placement to optimize accesses. Often, however, a system's size increases its complexity and it's susceptibility to unique failures such as correlated cascading failure.

Alia is designed with a single process model – the same process participating in the root quorum also handles messages for the subquorum(s) the process has been assigned to. This model ensures that if a replica fails it cannot participate in some decision making, such as configuration, but not others, such as accesses. This requirement also allows us to more easily tackle complex failures; such as using a disaster timeout to ensure progress even with a worst-case failure of delegates, or ensuring that leases are either respected or replaced for whole subquorums that fall out of communication.

Requirement 3: Consistency semantics must be transparent and interpretable. As privacy and security become increasingly important requirements of distributed systems, consistency is no longer about ensuring that your boss cannot see your Spring Break pictures on a social network wall. Instead, consistency is about ensuring that the correct operations are being executed on the correct replicas and that data can be audited to discover its exact placement. Alia ensures that there is an intersection between subquorums where data accesses are taking place and the root quorum where configuration and namespace partitions are occurring. This intersection is optimized by delegated voting to ensure that the root quorum can make progress and remain fluid. The intersection also guarantees that a complete, externalizable log of events for the global system can be exported on demand.

IX. Conclusion

The next generation of distributed systems will be geographically replicated around the planet in order to provide better performance by preventing bottlenecks and localizing accesses to international and highly mobile users and to provide durability in the face of catastrophic failure. We have presented hierarchical consensus and Alia, an extension and

implementation of Vertical Paxos, that are designed to scale coordination and transparently provide strong consistency in order to build and deploy systems that span globe. HC is a framework of intersecting tiers of quorums whose primary benefit is flexibility, which allows large systems to dynamically adapt to changing conditions, improving both performance and maintainability.

Hierarchical consensus handles challenges of geo-distributed consensus through flexible reconfiguration. Increasing network distance between replicas increases latency and the probability of network partitions, making strong consistency a challenge. To handle this, we separate the concerns of placement and access decisions to the root quorum and subquorums, allowing as much of the system to operate as independently as possible. Centralized administration is impossible in a global context, so the root quorum is able to adapt the system automatically by observing conditions and applying policy-driven changes to the system in real time with fuzzy transitions. To ensure correct reasoning of global consistency semantics and reduce the complexity of independent-subsystems with different failure modes, HC ensures that there is an intersection of the root quorum and subs. This intersection requires all nodes to participate in the root quorum, so to scale this quorum, we introduce delegated voting to improve globally availability. Finally, because objects have different requirements for availability or durability, data placement rules and subquorum behavior can be adjusted for different geographic access patterns.

REFERENCES

[1] CockroachDB Geo-Partitioning, 2020.

[2] David G Andersen, Jason Franklin, Michael Kaminsky, Amar Phanishayee, Lawrence Tan, and Vijay Vasudevan. FAWN: A fast array of wimpy nodes. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*, pages 1–14. ACM, 2009.

[3] Muthukaruppan Annamalai, Kaushik Ravichandran, Harish Srinivas, Igor Zinkovsky, Luning Pan, Tony Savor, David Nagle, and Michael Stumm. Sharding the shards: Managing datastore locality at scale with Akkio. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 445–460, 2018.

[4] Jason Baker, Chris Bond, James C. Corbett, J. J. Furman, Andrey Khorlin, James Larson, Jean-Michel Leon, Yawei Li, Alexander Lloyd, and Vadim Yushprakh. Megastore: Providing Scalable, Highly Available Storage for Interactive Services. In *CIDR*, volume 11, pages 223–234, 2011.

[5] Mahesh Balakrishnan, Dahlia Malkhi, Ted Wobber, Ming Wu, Vijayan Prabhakaran, Michael Wei, John D Davis, Sriram Rao, Tao Zou, and Aviad Zuck. Tango: Distributed data structures over a shared log. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 325–340. ACM, 2013.

[6] Martin Biely, Zoran Milosevic, Nuno Santos, and Andre Schiper. S-paxos: Offloading the leader for high throughput state machine replication. In *Reliable Distributed Systems (SRDS), 2012 IEEE 31st Symposium On*, pages 111–120. IEEE, 2012.

[7] Mike Burrows. The Chubby lock service for loosely-coupled distributed systems. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, pages 335–350. USENIX Association, 2006.

[8] Lásaro Jonas Camargos, Rodrigo Malta Schmidt, and Fernando Pedone. Multicoordinated paxos. In *Proceedings of the Twenty-Sixth Annual ACM Symposium on Principles of Distributed Computing*, pages 316–317. ACM, 2007.

[9] Brian F. Cooper, Raghu Ramakrishnan, Utkarsh Srivastava, Adam Silberstein, Philip Bohannon, Hans-Arno Jacobsen, Nick Puz, Daniel Weaver, and Ramana Yerneni. PNUTS: Yahoo!'s hosted data serving platform. 1(2):1277–1288, 2008.

[10] James C. Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, J. J. Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, et al. Spanner: Google's globally distributed database. 31(3):8, 2013.

[11] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: Amazon's highly available key-value store. In *ACM SIGOPS Operating Systems Review*, volume 41, pages 205–220. ACM, 2007.

[12] Lisa Glendenning, Ivan Beschastnikh, Arvind Krishnamurthy, and Thomas Anderson. Scalable consistency in Scatter. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, pages 15–28. ACM, 2011.

[13] H. Howard, D. Malkhi, and A. Spiegelman. Flexible Paxos: Quorum intersection revisited. 2016-08.

[14] Heidi Howard, Malte Schwarzkopf, Anil Madhavapeddy, and Jon Crowcroft. Raft refloated: Do we have consensus? 49(1):12–21, 2015.

[15] Patrick Hunt, Mahadev Konar, Flavio Paiva Junqueira, and Benjamin Reed. ZooKeeper: Wait-free Coordination for Internet-scale Systems. In *USENIX Annual Technical Conference*, volume 8, page 9, 2010.

[16] Sudarshan Kadambi, Jianjun Chen, Brian F. Cooper, David Lomax, Raghu Ramakrishnan, Adam Silberstein, Erwin Tam, and Hector Garcia-Molina. Where in the world is my data. In *Proceedings International Conference on Very Large Data Bases (VLDB)*, 2011.

[17] Matt Klein. Lyft's Envoy: Experiences Operating a Large Service Mesh. 2017.

[18] Tim Kraska, Gene Pang, Michael J. Franklin, Samuel Madden, and Alan Fekete. MDCC: Multi-Data Center Consistency. In *Proceedings of the 8th ACM European Conference on Computer Systems*, pages 113–126. ACM, 2013.

[19] Leslie Lamport. Paxos made simple. 32(4):18–25, 2001.

[20] Leslie Lamport. Generalized consensus and Paxos, 2005.

[21] Leslie Lamport. Fast paxos. 19(2):79–103, 2006.

[22] Leslie Lamport, Dahlia Malkhi, and Lidong Zhou. Vertical paxos and primary-backup replication. In *Proceedings of the 28th ACM Symposium on Principles of Distributed Computing*, pages 312–313. ACM, 2009.

[23] John MacCormick, Chandramohan A. Thekkath, Marcus Jager, Kristof Roomp, Lidong Zhou, and Ryan Peterson. Niobe: A practical replication protocol. 3(4):1, 2008.

[24] Yanhua Mao, Flavio Paiva Junqueira, and Keith Marzullo. Mencius: Building efficient replicated state machines for WANs. In *OSDI*, volume 8, pages 369–384, 2008.

[25] Iulian Moraru, David G. Andersen, and Michael Kaminsky. There is more consensus in egalitarian parliaments. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 358–372. ACM, 2013.

[26] Diego Ongaro and John Ousterhout. In search of an understandable consensus algorithm. In *2014 USENIX Annual Technical Conference (USENIX ATC 14)*, pages 305–319, 2014.

[27] Aashaka Shah, Vinay Banakar, Supreeth Shastri, Melissa Wasserman, and Vijay Chidambaram. Analyzing the Impact of GDPR on Storage Systems. In *11th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 19)*, 2019.

[28] Alexander Thomson and Daniel J. Abadi. CalvinFS: Consistent wan replication and scalable metadata management for distributed file systems. In *13th USENIX Conference on File and Storage Technologies (FAST 15)*, pages 1–14, 2015.

[29] Robbert Van Renesse and Fred B Schneider. Chain Replication for Supporting High Throughput and Availability. In *OSDI*, volume 4, pages 91–104, 2004.

[30] Alexandre Verbitski, Anurag Gupta, Debanjan Saha, Murali Brahmadesam, Kamal Gupta, Raman Mittal, Sailesh Krishnamurthy, Sandor Maurice, Tengiz Kharatishvili, and Xiaofeng Bao. Amazon aurora: Design considerations for high throughput cloud-native relational databases. In *Proceedings of the 2017 ACM International Conference on Management of Data*, pages 1041–1052. ACM, 2017.

[31] Michael Wei, Amy Tai, Christopher J Rossbach, Ittai Abraham, Maithem Munshed, Medhavi Dhawan, Jim Stabile, Udi Wieder, Scott Fritchie, Steven Swanson, et al. vCorfu: A Cloud-Scale Object Store on a Shared Log. In *NSDI*, pages 35–49, 2017.